Operating Systems

Lecture 1.6

Understanding Process Stack and Heap
Behind the Curtain

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

- Users, Programs and Processes
- Life Cycle of a Process
- Process model in Linux
- Program control block
- Command Line Arguments
- Environment Variables
- Function Calling Convention (FSF)
- Process Heap

Users, Programs and Processes

Users, Programs and Processes

Users:

- In modern operating systems like Linux, Windows and Mac, the system is designed to support multiple users simultaneously.
- Each user is assigned a distinct account that grants access to personal storage, permissions, and execution privileges.
- Users can create, compile, and store a wide variety of programs, ranging from shell scripts to compiled executables, stored on the system's disk.
- These stored programs are persistently available and can be executed when needed.
- The operating system provides process isolation and concurrency control, enabling multiple users to execute the same program independently and simultaneously.
- Additionally, a single user may run multiple instances of the same program in parallel. This architecture promotes both efficient resource utilization and secure, collaborative computing within the shared environment of the operating system.

Users, Programs and Processes (cont...)

Programs:

- A *program* is a passive entity composed of a set of instructions written to perform a specific task.
- Examples include executable files (`a.out`), scripts (`script.sh`), or applications (`calculator.exe`, `chrome.app`).
- These programs are stored on secondary storage in well-defined formats, and they remain static and inactive until explicitly invoked.
- Programs are stored on disk, and consume no system resources (CPU cycles, memory, or I/O bandwidth).
- They serve as templates for execution, waiting in a dormant state until a user or another process initiates them.
- When a program is launched, the operating system takes responsibility for loading it into main memory and preparing it for execution.

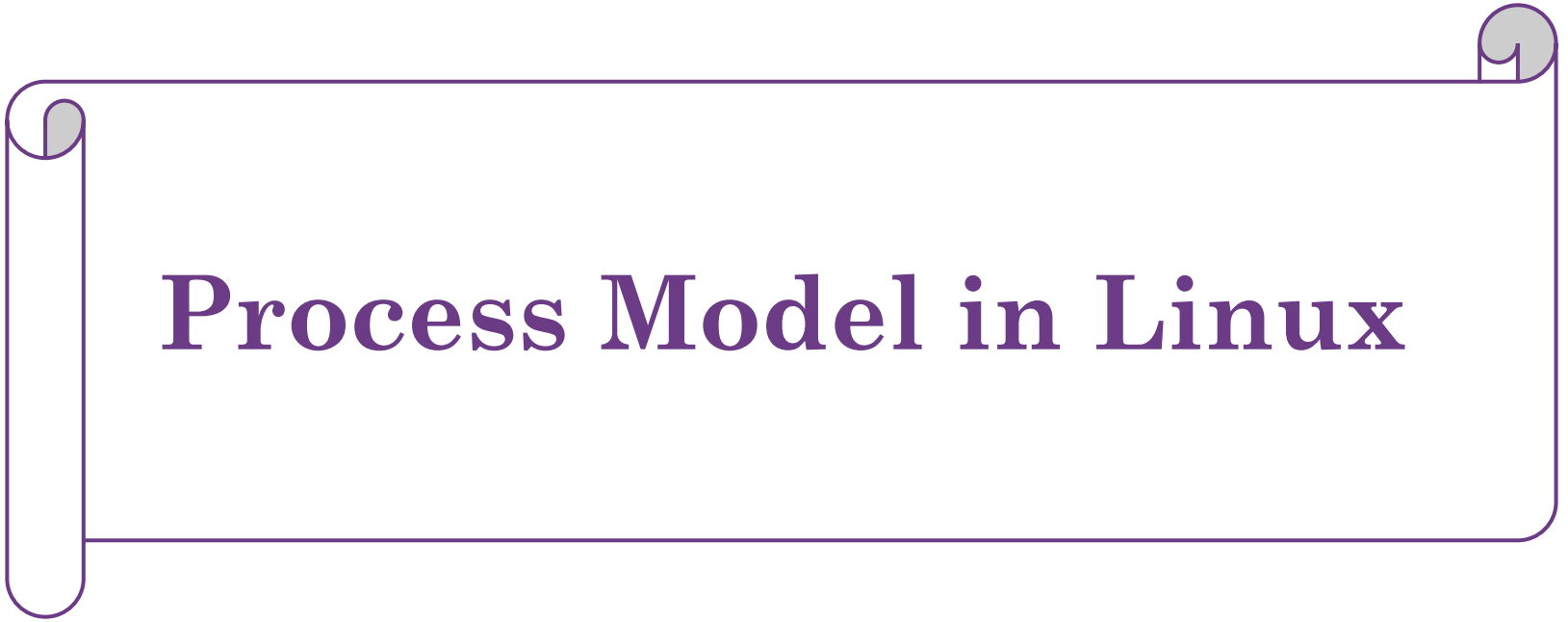
Users, Programs and Processes (cont...)

Processes:

- A ***process*** is an active, runtime instance of a program (a program in execution).
- Unlike a program, which is a passive file on disk, a process is a dynamic entity that occupies system resources.
- When a user executes a program, the operating system loads its code into RAM, allocates memory, assigns a unique process identifier (PID), and establishes required structures such as the process control block (PCB).
- The CPU can then schedule this process for execution.
- During its lifetime, a process utilizes CPU time, accesses system memory, and may perform I/O operations such as reading from or writing to files, interacting with other devices, or communicating over the network.
- For example, when the calculator application is launched, the system creates a calculator process that actively consumes resources to perform calculations as requested.

Lifecycle of a Process?

1. **Program Creation:** The life cycle begins when a user writes source code in a programming language such as C or Python. This source code is then compiled (in the case of compiled languages) or saved directly (for interpreted languages) to produce an executable file that is stored on system's secondary storage in a specified binary format.
2. **Program Storage:** The executable file resides on disk as a passive entity, as it does not consume any system resources (CPU time, memory). The program remains dormant, waiting for a user or another process to initiate its execution.
3. **Process Creation:** When a user decides to run the program, (via a command-line instruction, GUI action, or script) the operating system initiates *process creation*. This involves several internal OS mechanisms: the program code is loaded into main memory (RAM), a unique Process ID (PID) is assigned, and a *Process Control Block (PCB)* is created to track the process's state, resources, and metadata. At this point, the program becomes an active entity called *process* under OS management.
4. **Process Execution:** Once created, the process is scheduled by the *CPU scheduler* for execution. The instructions within the program are fetched, decoded, and executed by the CPU. During this phase, the process may allocate memory, perform file I/O, send or receive data over the network, or interact with other system resources. The OS ensures resource isolation and manages context switching between multiple processes to enable multitasking.
5. **Process Termination:** Eventually, the process reaches completion, either by successfully finishing its task (normal termination), encountering an error (abnormal termination), or being forcibly terminated by the user or another process. Upon termination, the operating system reclaims all resources allocated to the process, including memory, file handles, and process table entries.



Process Model in Linux

Program Control Block

A Process Control Block (PCB) is a data structure used by the operating system to manage information about a process. In Linux, PCB is implemented as the `task_struct` structure, which resides in kernel space and encapsulates all critical information required to manage a process. This structure includes the process ID, current state, scheduling information, memory management details, open file descriptors, and CPU register context for context switching. By maintaining this centralized data structure for every process, the kernel efficiently supports scheduling, resource management, inter-process communication, and process lifecycle control.

Process Identification

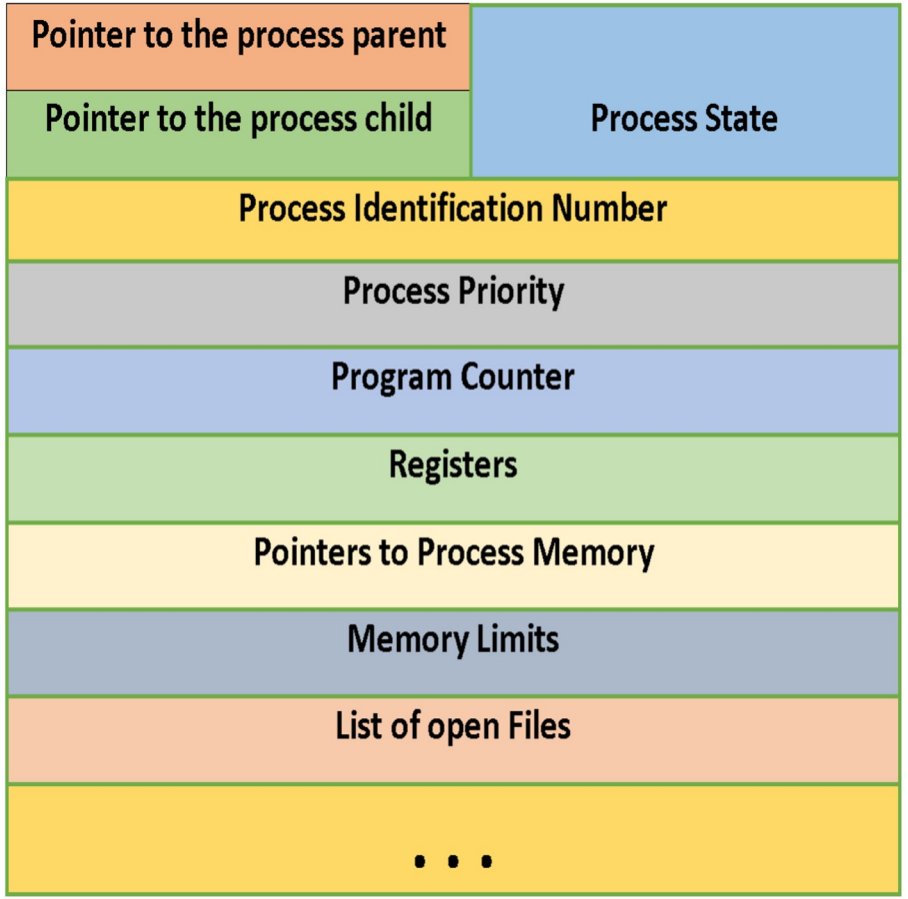
- PID & PPID
- UID & GID
- Saved SUID & SGID
- File System UID & GID

Process state information

- User Visible Registers
- Control and Status Registers (flags)

Process control information

- Scheduling Info
- Privileges Info
- Memory Management Info
- Resource Ownership and Utilization

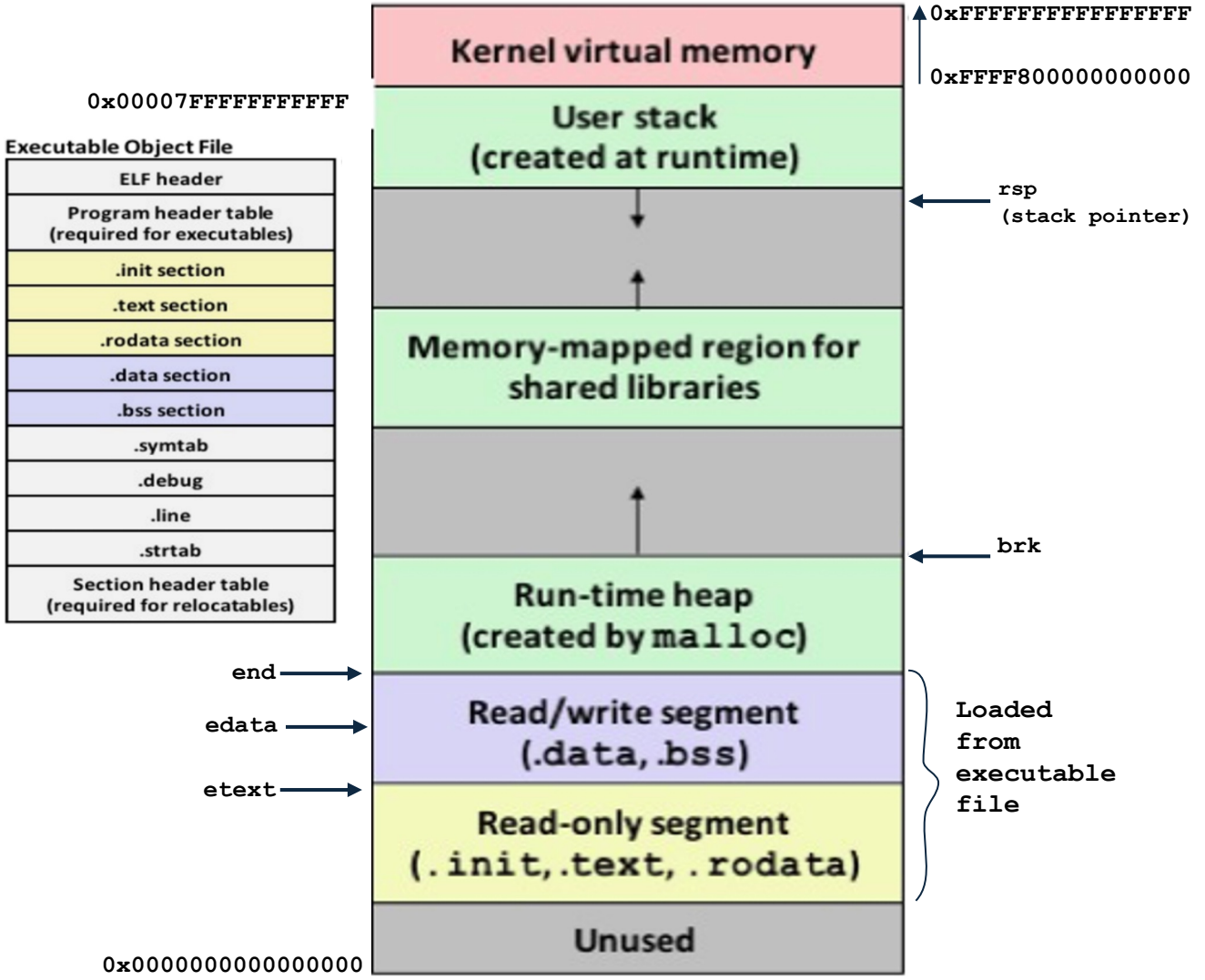


64-bit ELF to Process Logical Address Space

Program Loading is a process of copying a program from disk to main memory in order to make it a process

On x86-64 Linux systems, memory addresses must follow the *canonical form*. If bit 47 is 0, bits 48 through 63 must also be 0, forming valid user-space addresses. If bit 47 is 1, bits 48 through 63 must also be 1, forming valid kernel-space addresses. Any violation of this rule results in a *general protection fault*, as the address falls into the invalid *non-canonical region*.

- **User space** starts from `0x0000000000000000` to `0x00007FFFFFFFFFFF`, providing 128 TiB of addressable memory. This region is accessible to user-mode processes and is where the code, data, heap, and stack reside.
- **Kernel space** starts from `0xFFFF800000000000` to `0xFFFFFFFFFFFFFFFF`, spanning 128 TiB. It is exclusively reserved for the operating system kernel, its subsystems, device drivers, and essential kernel data structures. Only code running in privileged mode (ring 0) can access this region. Attempts to access it from user mode result in a protection fault, ensuring kernel integrity and isolation.
- **The non-canonical space**, ranging from `0x0008000000000000` to `0xFFF7FFFFFFFFFFFF`, lies between the user and kernel address ranges (16 EiB minus 256 TiB).



Critical System Variables:

- **`etext`**: Points to the first address above the code/text section
- **`edata`**: Points to the first address above the initialized data section (`.data`)
- **`end`**: Points to the first address above the `.bss` section
- **`brk`**: Points to the current top of the heap (initially a little above `end`)

Demonstration


```
$ sudo echo 0 | tee /proc/sys/kernel/randomize_va_space
```

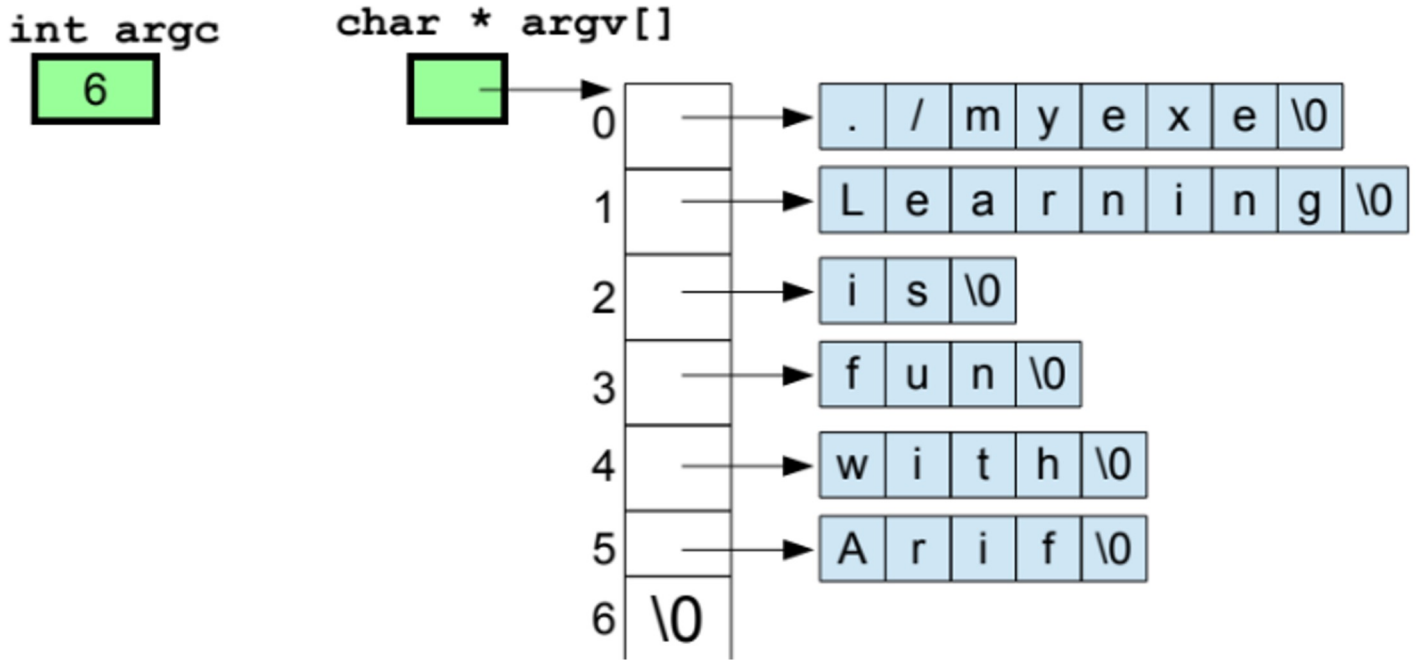
GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Command Line Arguments

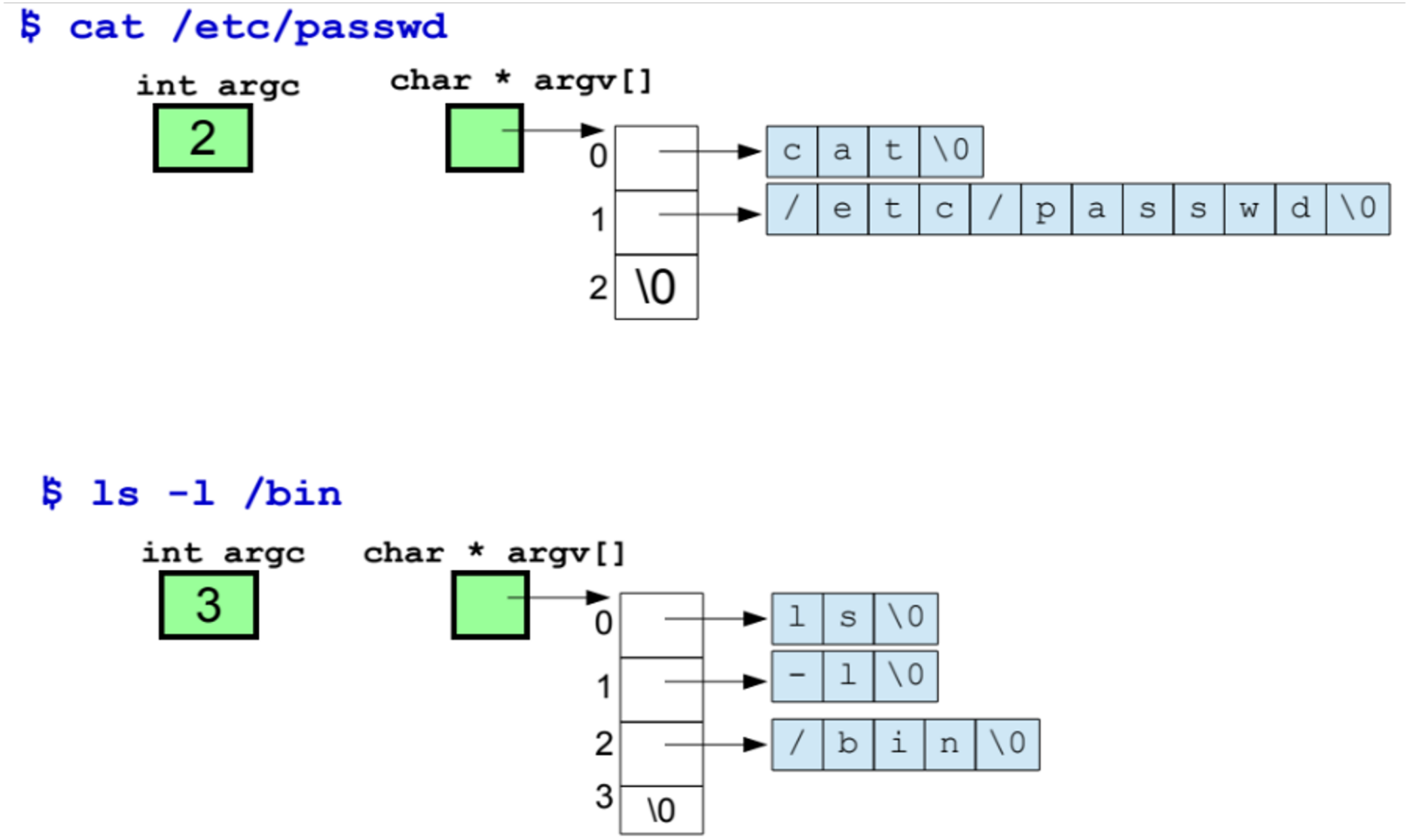
Command Line Arguments

```
int main(int argc, char *argv[]){
   printf("No of arguments passed are: %d\n",argc);
   printf("Parameters are:\n");
   for(int i = 0; argv[i] != NULL ; i++)
      printf("argv[%d]:%s \n", i, argv[i]);
   return 0;
}
```

```
$ ./myexe Learning is fun with Arif
```



Use of Command Line Arguments



100$ Question: Why program name is passed to the process as `argv[0]`?

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Environment Variables

What are Environment Variables?

Environment variables are name-value pairs. Each running process has a block of memory that contains a set of the name-value pairs which usually come from its parent process. When you run a command inside a Linux shell, the shell program (parent) send its own environment variables along with some new environment variables to the child process. These environment variables sit in the memory of the child process, and if the child process does not use these variables at all, then these variables will have no impact on its execution. But if the child process uses these variables, then these variables will of course have an impact on its behavior.

Variable	Description
PATH	Specifies the directories where the system looks for executable files
HOME	Points to the current user's home directory
USER	The current logged in user / Contains the username of the current user
PWD	The current working directory
SHELL	Specifies the default shell for the user
TERM	Different hardware terminals can be emulated. It displays the current terminal type (for example, xterm)
HISTSIZE	The maximum number of lines of command history allowed to be stored in memory
HISTFILE	File where command history is saved when shell exits
HISTFILESIZE	the maximum number of lines contained in the history file
PS1, PS2	The default prompt in bash, and Secondary prompt string used for multi-line commands
MANPATH	Directories to search for manual pages
LD_LIBRARY_PATH	Directories to search for shared libraries
LD_PRELOAD	Specify one or more shared libraries that should be loaded before any other libraries
IFS	Internal Field Separator used by shell for word splitting
PPID	Process ID of parent process
$$	Process ID of current shell

Playing with Environment Variables

- We can use environment variables for personalization and configuration of our system. It is similar to customizing Windows desktop, screensaver, and default browser.
- UNIX systems store user preferences in environment variables. Each user has unique settings for their computing environment.
- One can use following shell commands to display environment variables:

```
$ env                - Display only environment variables
$ set                - Display environment, shell variables as well as functions
$ echo $PATH         - Display specific variable value
```

- One can use following shell command create a new or change value of an existing variable:

```
$ PATH=$PATH:/tmp/abc/
```

- The above command will make a temporary change to the PATH environment variable that will persist for the current session only. To make it persistent for a specific user, we need to enter the name-value pair inside the `~/.bashrc` or `~/.bash_profile` file. To create a persistent environment variable we need to edit the `/etc/profile` file. Once done to make the changes active, we need to `source` the file as shown:

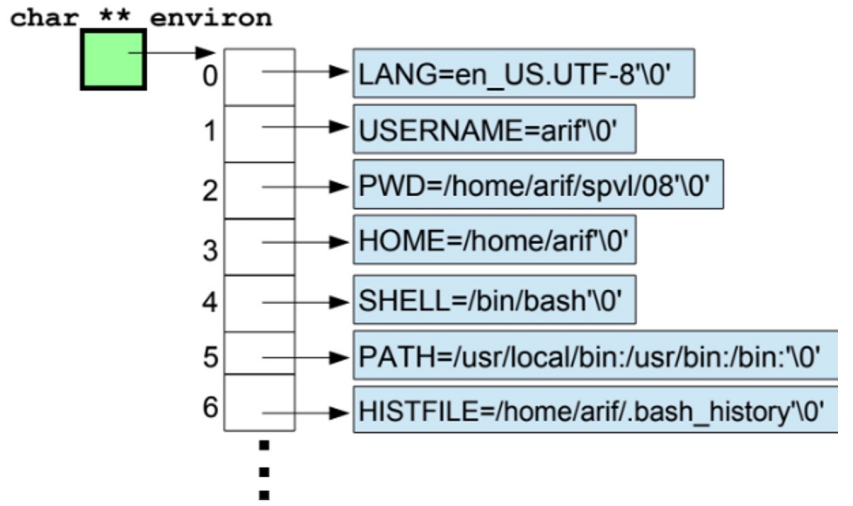
```
$ source ~/.bashrc
```

- Whenever we execute a program or command from the shell, a copy of these environment variables is passed to that program for proper execution. For details read the manual page of `environ`.
- Use `export` to ensure the created variable is inherited by child processes. Assignment w/o export creates a shell-local. Variable not visible to sub-processes:

```
$ export myname="Arif Butt"
```

Accessing Environment Variables in C Program

- **Method 1:** Use the `char *getenv(const char *name)` function to retrieve specific environment variable by name, that returns a pointer to value string or NULL if not found. Reflects changes made by `setenv()/putenv()`
- **Method 2:** Use the `extern char** environ` variable to access entire environment array, and then iterate through all environment variables. It returns an array of "name=value" NLL terminated strings. Reflects changes made by `setenv()/putenv()`
- **Method 3:** Use the third argument to `main()`, i.e., `char* envp[`. It is quite similar to Method 2, but do not reflect runtime changes to environment made by `setenv()/putenv()`



Method 3

```
#include <stdio.h>
int main(int argc, char *argv[], char *envp[])
{
    int i = 0;
    printf("Environment variables:\n");
    while (envp[i] != NULL) {
        printf("%s\n", envp[i]);
        i++;
    }
    return 0;
}
```

Method 2

```
#include <stdio.h>
extern char **environ;
int main() {
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
    return 0;
}
```

Method 1

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char *home = getenv("HOME");
    char *user = getenv("USER");
    if (home != NULL)
        printf("Home dir: %s\n", home);
    if (user != NULL)
        printf("Username: %s\n", user);
    return 0;
}
```

Modifying Environment Variables in C Program

- The way we can change environment variables on the shell, we can also change them form within a C program, as well as can create a new environment variable using library functions shown below.
- We create new or set existing environment variables to build a suitable environment for a process to run. Moreover, it is considered a form of Inter Process Communication, as a child gets a copy of its parent's environment variables at the time it is created.

```
char *getenv(const  char *name);
int putenv(char *string);
int setenv(char *name ,char *val, int overwrite);
int unsetenv(const char *name);
int clearenv();
```

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    setenv("var1", "Hello World", 1);
    putenv("var2=Hello Arif");
    char *value = getenv("var1");
    printf("var1 = %s\n", value);
    unsetenv("var1");
    return 0;
}
```

Demonstration

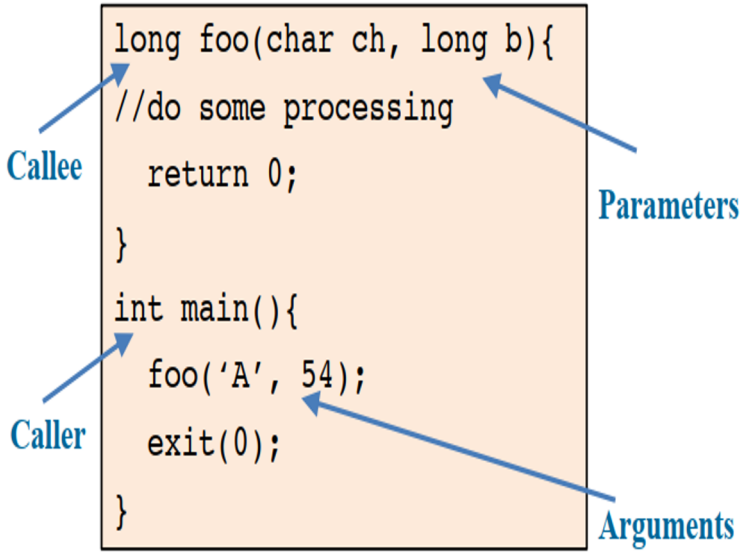
GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*


Function Calling Convention
&
Function Stack Frame

Function Calling Convention

The **function calling convention** is a set of rules that dictate *how functions receive parameters, return values, manage the stack, and how to share the CPU registers between the caller and the callee.*

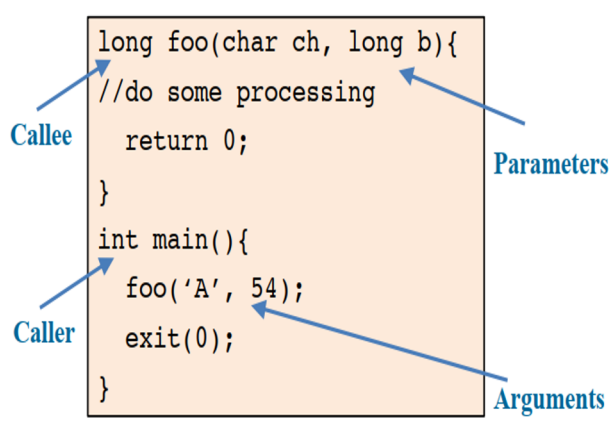
- How the function arguments are passed?
  - Through the stack
  - Using CPU Registers
  - A combination of both (typically registers for the first few arguments, then stack)
- In what order are arguments passed?
  - Right to left (common in many C conventions)
  - Left to right (used in some specific conventions or languages)
- Who is responsible for creating the FSF?
  - The Callee (the function being called)
  - The Caller (the function making the call)
- Who is responsible for unwinding the stack?
  - The Callee (the function being called)
  - The Caller (the function making the call)



Function Calling Convention (cont...)

In high level programming languages like C and C++, the values passed by the caller to the callee are called arguments. When the values are received by the called subroutine, they are called parameters. In programming, the terms "caller" and "callee" refer to the relationship between functions or procedures in the context of function calls:

- Caller Function: The function that initiates a call to another function to perform a task or compute a result.
- Callee Function: This is the function being called by another function. It is the one that gets executed as a result of the call.

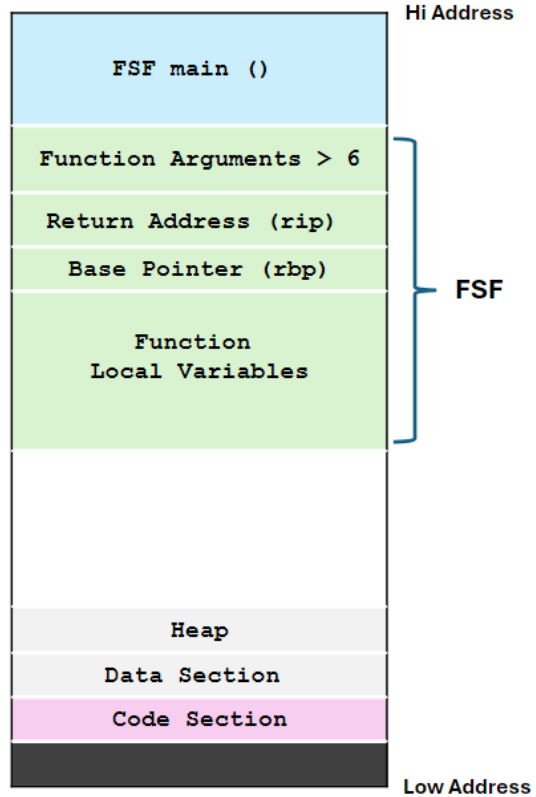


In the 16-bit and 32-bit days, since there were only eight general purpose registers in x-86 architecture, therefore, all the arguments were passed by the caller to the callee by pushing the arguments on the stack. On x86-64 processor, Linux, Solaris and Mac Operating Systems use a function call protocol called the System-V AMD64 ABI. In which first six integer parameters are passed via registers: **`rdi`**, **`rsi`**, **`rdx`**, **`rcx`**, **`r8`**, **`r9`**, and first eight floating point parameters via **`xmm0`** to **`xmm7`** registers (rest on the runtime stack). On the contrary MS Windows Operating System use MS X64 Calling Convention, in which first four integer parameters are passed via registers: **`rcx`**, **`rdx`**, **`r8`**, **`r9`**, and first four floating point parameters via **`xmm0`** to **`xmm3`** registers (rest on the runtime stack). Both Linux and MS Windows use **`rax`** register to return integer values and **`xmm0`** register to return floating point values.

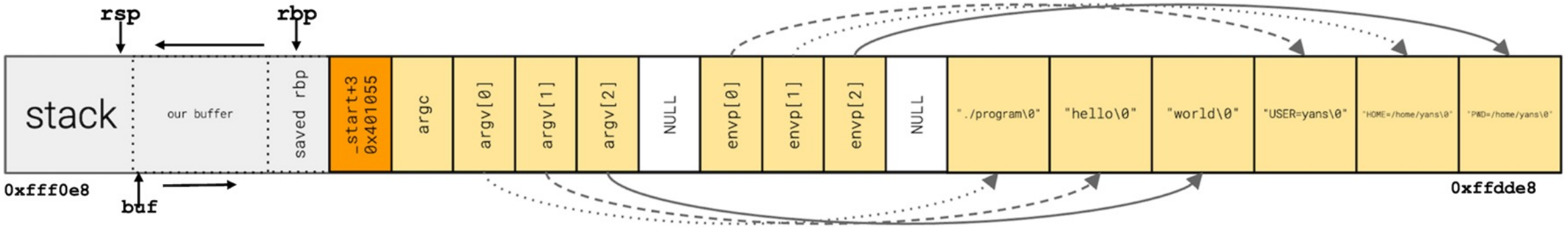
Layout of Process Stack (FSF)

The figure on the right, shows an abstract level view of Function Stack Frame (FSF) of a function called by `main()`, containing following four items: :

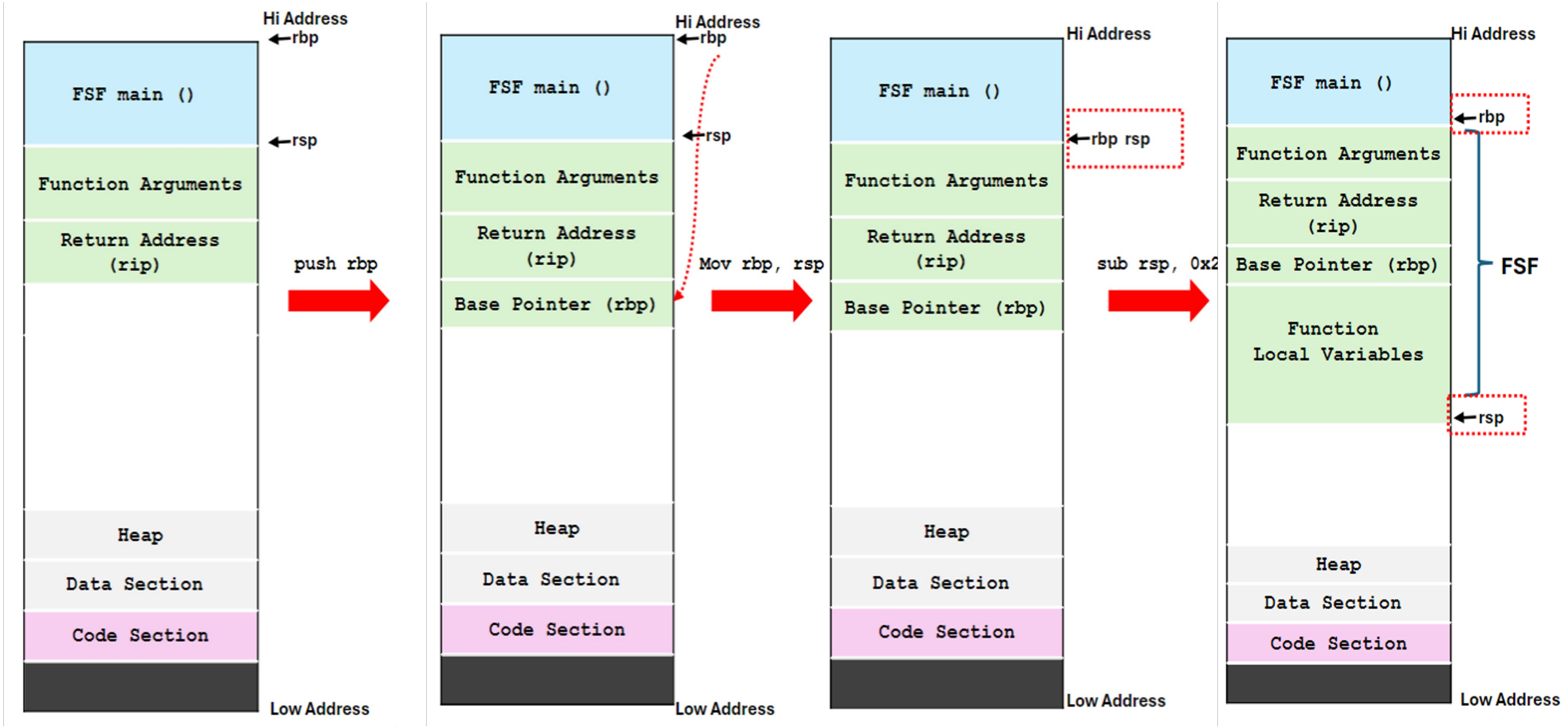
- Function Arguments (>6)
- Return Address
- Base Pointer (`rbp`) used to access variables at fixed offsets
- Local Variables



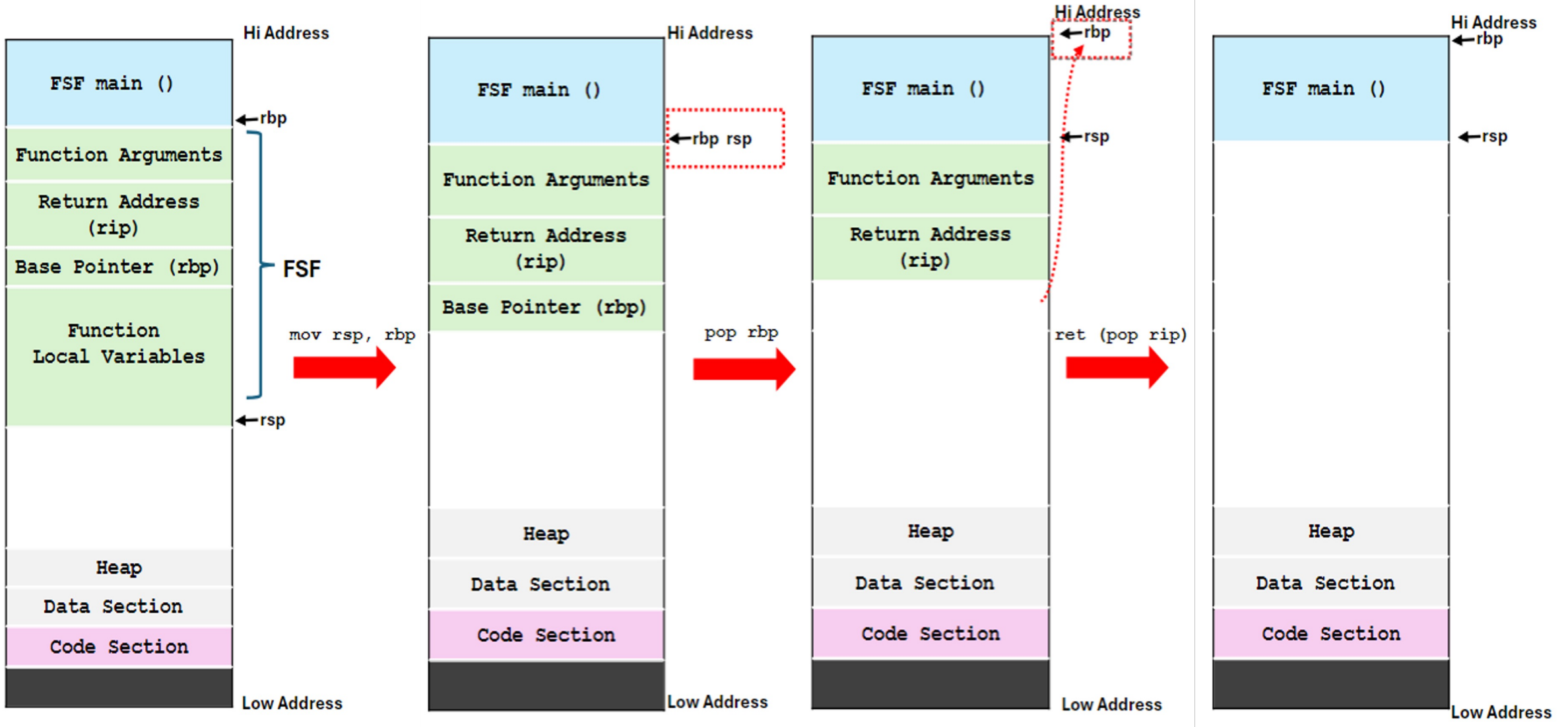
The figure below, shows a horizontal view of a process stack, which is executed by the shell (`$ ./program hello world`), displaying the command line arguments and environment variables in the FSF of `main()` function.



Stack Growing (Function Prologue)



Stack Shrinking (Function Epilogue)

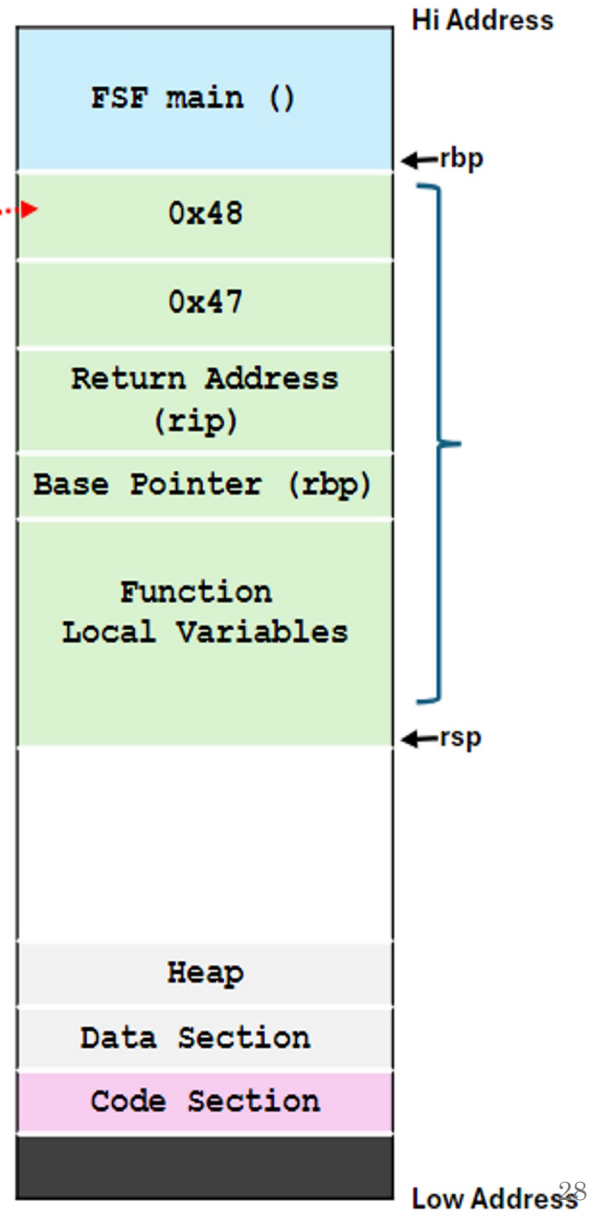


Concrete Example

```
long foo(long a, long b, long c, long d, long e, long f, long
g, long h){
//some computation is done
   return 1;
}

int main(){
long rv =foo(0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48);
  return rv;
}
```

r9:	0x46
r8:	0x45
rcx:	0x44
rdx:	0x43
rsi:	0x42
rdi:	0x41



Understanding Calling Convention with gdb (GEF)

Debugging C Program inside GEF

Let us now debug the following C program. In the source file, the `main()` function creates two long variables `main_var1` and `main_var2` and character pointer `*main_str2` and calls a function `f1()` and passing 8 parameters to that function. The function `f1()` receives 8 parameters and further creates two local variables and then calls another function `f2()` and passes one parameter to it. The `f2()` function receives a single a parameter, performs some operations and returns a value to `f1()` that further returns 1 to parent function which is `main()` and finally `main()` returns 0 to its parent which is the shell program.

When you run gdb with gef you get the following prompt, where you can give the `gef` command to view brief description of different `gef` commands:

```
gef> gef
```

Let us load the binary named `debugme`, set a breakpoint at `main`, and run the program step by step to understand the calling convention:

```
gef> file ./func-calling
gef> gef config context.layout "regs stack code"
gef> break main
gef> run
```

Do observe all the calling conventions concepts discussed in previous slides to have a crystal clear understanding with a hands on experience.

```
//Lec-1.6/func-calling-convention/func-calling.c
#include <stdio.h>
#include <stdlib.h>
   int f2(int a){
   int b = a +1;
   return b;
}

int f1(long a, long b, long c, long d, long e, long f, long g, long h){
   unsigned long f1_var1 = 0x123456789;
   unsigned long f1_var2 = 0x0abcdef;
   int rv = f2(5);
   return 1;
}

int main(int argc, char *argv[]){
   unsigned long main_var1 = 0x1122334455667788;
   unsigned long main_var2 = 0x99aabbccddeeff00;
   char *main_str2  = "Arif";
   int rv_f1 = f1(0x11111111, 0x22222222, 0x33333333, 0x44444444, 0x55555555,
0x66666666, 0x77777777, 0x88888888);
   return 0;
}
```

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Typical Stack Misuses and Errors

- **Stack Buffer Overflow:** Writing beyond the allocated boundaries of a stack-based buffer, typically through unsafe functions like `strcpy()`, `gets()`, or `sprintf()`. This corrupts adjacent stack data including local variables, saved frame pointers, and return addresses, potentially allowing arbitrary code execution.
- **Off-by-One Errors:** A subtle form of buffer overflow where loops or bounds checks are off by exactly one position (e.g., `for(i=0; i<=size; i++)`), overwriting stack metadata like canaries or frame pointers.
- **Stack Underflow:** Accessing memory below a buffer's base address through negative array indices or incorrect pointer arithmetic.
- **Use of Uninitialized Variables:** Reading stack variables before assignment, leading to unpredictable behavior since stack memory contains residual data from previous function calls. This creates information disclosure risks and logic errors.
- **Return Address Overwrites:** Deliberately corrupting the saved return address on the stack to redirect program execution. Modern variants involve Return-Oriented Programming (ROP) or Jump-Oriented Programming (JOP) to bypass security mitigations.
- **Stack Exhaustion/Overflow:** Depleting available stack space through unbounded recursion, excessively large local arrays, or variable-length arrays (VLAs) with untrusted size parameters. This triggers segmentation faults or stack guard page violations.
- **Format String Vulnerabilities:** Using untrusted input as format strings in `printf()` family functions, allowing attackers to read from or write to arbitrary stack locations using format specifiers like %n.

These vulnerabilities are mitigated through compiler protections (stack canaries, ASLR, DEP/NX bit), safe coding practices, and static analysis tools.

Compiler Explorer

https://godbolt.org/

COMPILER EXPLORER Add... More Templates

Sponsors intel Cppcon Google Share

C source #1

Save/Load + Add new... Vim C

```c
#include <stdio.h>
#include <stdlib.h>

int foo(int a, int b, int c, int d, int e, int f, int g, int h){
   return 1;
}

int main(int argc, char *argv[]){
   int rv = foo(1, 2, 3, 4, 5, 6, 7, 8);
   return 0;
}
```

x86-64 gcc 14.3 (Editor #1)

x86-64 gcc 14.3 -O0 -std=c18 -m64

Output... Filter... Libraries Overrides + Add new... Add tool...

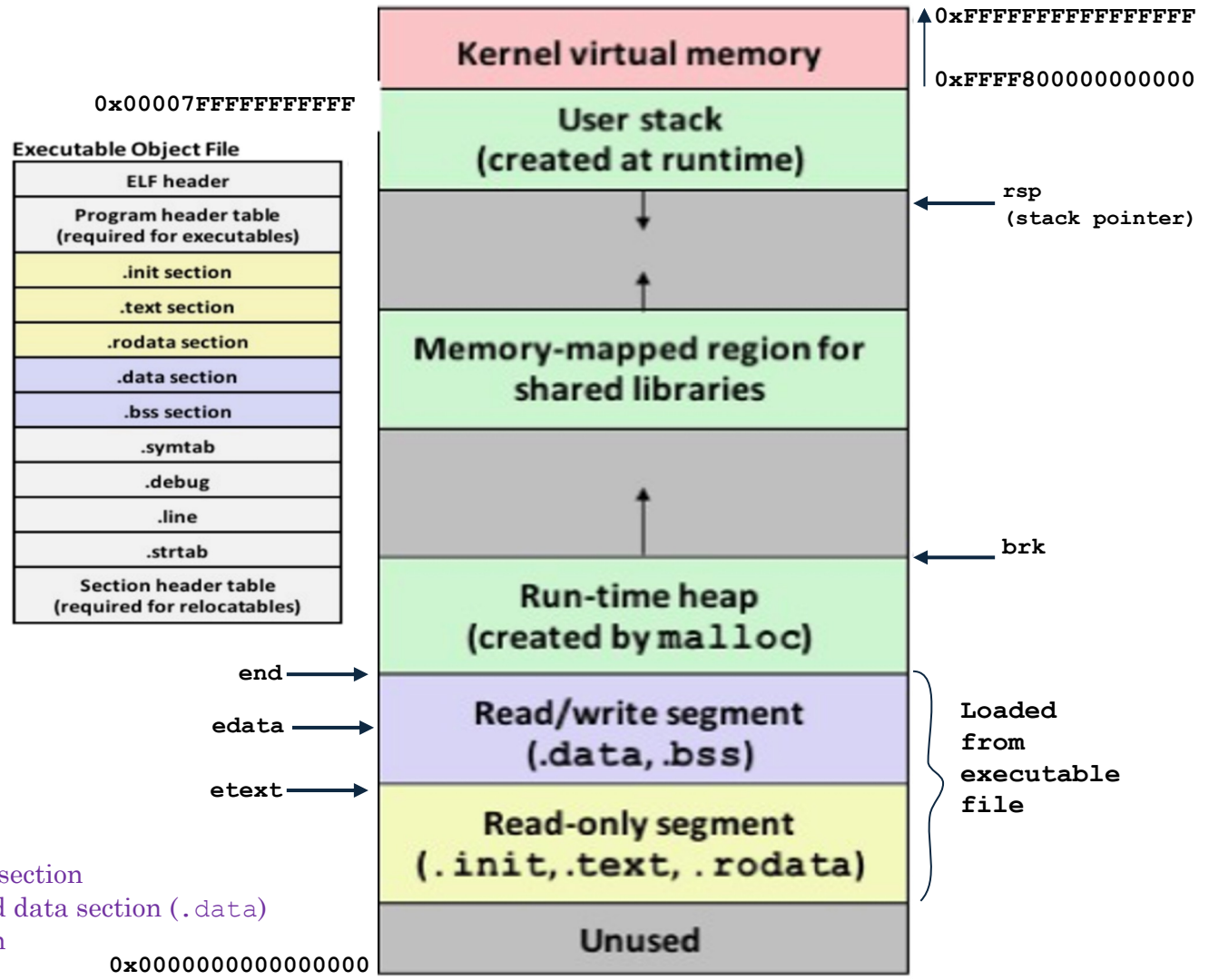
```asm
foo:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     DWORD PTR [rbp-16], ecx
        mov     DWORD PTR [rbp-20], r8d
        mov     DWORD PTR [rbp-24], r9d
        mov     eax, 1
        pop     rbp
        ret
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR [rbp-20], edi
        mov     QWORD PTR [rbp-32], rsi
        push    8
        push    7
        mov     r9d, 6
        mov     r8d, 5
        mov     ecx, 4
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    foo
        add     rsp, 16
        mov     DWORD PTR [rbp-4], eax
        mov     eax, 0
        leave
        ret
```


Output (0/0) x86-64 gcc 14.3 - 4540ms (6407B) ~411 lines filtered Compiler License

Process Heap

Overview of Process Heap Memory

- Heap is a memory segment for runtime allocation when memory requirements are unknown at compile time.
- Heap is process-wide accessible memory region, unlike stack memory which is function-scoped.
- Unlike the stack, which operates in a strict Last-In-First-Out (LIFO) manner with fixed size and automatic memory management, the heap provides complete flexibility in allocation patterns but requires explicit memory management through `malloc/free` or `new/delete` operations.
- At a low level, dynamic memory functions are backed by system calls such as `brk()` and its deprecated wrapper `sbrk()` to expand or contract the process's heap segment. Modern allocators may also use `mmap()` to request memory directly from the operating system for large or page-aligned allocations.
- The heap allocator (e.g., glibc's `ptmalloc`) typically requests memory in page-sized chunks (often 4 KB or more), which are then subdivided and managed internally.



Critical System Variables:

- **`etext`**: Points to the first address above the code/text section
- **`edata`**: Points to the first address above the initialized data section (`.data`)
- **`end`**: Points to the first address above the `.bss` section
- **`brk`**: Points to the current top of the heap

Heap Allocators

Dynamic memory allocation in the heap is facilitated by **memory allocators**, which are software components responsible for managing, allocation and deallocation of memory blocks. Allocators generally fall into two primary categories, based on how memory is reclaimed:

Explicit Allocators

- In explicit allocation, the application code is responsible for both allocating and freeing memory.
- The allocator provides APIs to allocate memory (e.g., `malloc` in C or `new` in C++) and to deallocate it (e.g., `free` in C or `delete` in C++). Other languages that support explicit allocators are Fortran, Pascal, Ada, Rust, and assembly language.
- This model offers fine-grained control and predictable performance, but it places the burden of memory management on the programmer.
- Common problems associated with explicit allocators include:
  - Memory leaks (forgetting to free)
  - Dangling pointers (accessing memory after it has been freed)
  - Double frees (freeing the same block multiple times)

```
void *p = malloc(100);
/*...*/
free(p);
```

```
int *arr = new int[10];
/*...*/
delete[] arr;
```

Implicit Allocators (Garbage Collectors)

- Implicit allocation still requires the application to explicitly request memory, but the allocator automatically detects and reclaims unused memory.
- This detection and cleanup process is known as garbage collection.
- Garbage collectors track object references at runtime and reclaim memory that is no longer reachable by the application.
- This model improves programmer productivity and memory safety, at the cost of runtime overhead and less predictable performance.
- Languages such as Java, JavaScript, Python, and Lisp rely on garbage collection to free allocated blocks.

Allocating and freeing memory on heap

```
void *malloc (size_t size);

void *calloc(size_t noOfObjects, size_t size);

void *realloc (void* ptr, size_t newsize);
```

- **`malloc()`** allocates size bytes from the heap and returns a pointer to the start of the newly allocated block of memory. On failure returns `NULL` and sets errno to indicate error.
- **`calloc()`** allocates space for specific number of objects, each of specified size. Returns a pointer to the start of the newly allocated block of memory. Unlike `malloc()`, `calloc()` initializes the allocated memory to zero. On failure returns `NULL` and sets `errno` to indicate error.
- **`realloc()`** is used to resize a block of memory previously allocated by one of the functions in `malloc()` package. `ptr` argument is the pointer to the block of memory that is to be resized. On success `realloc()` returns a pointer to the location of the resized block, which may be different from its location before the call. On failure, returns `NULL` and leaves the previous block pointed to by pointer untouched.

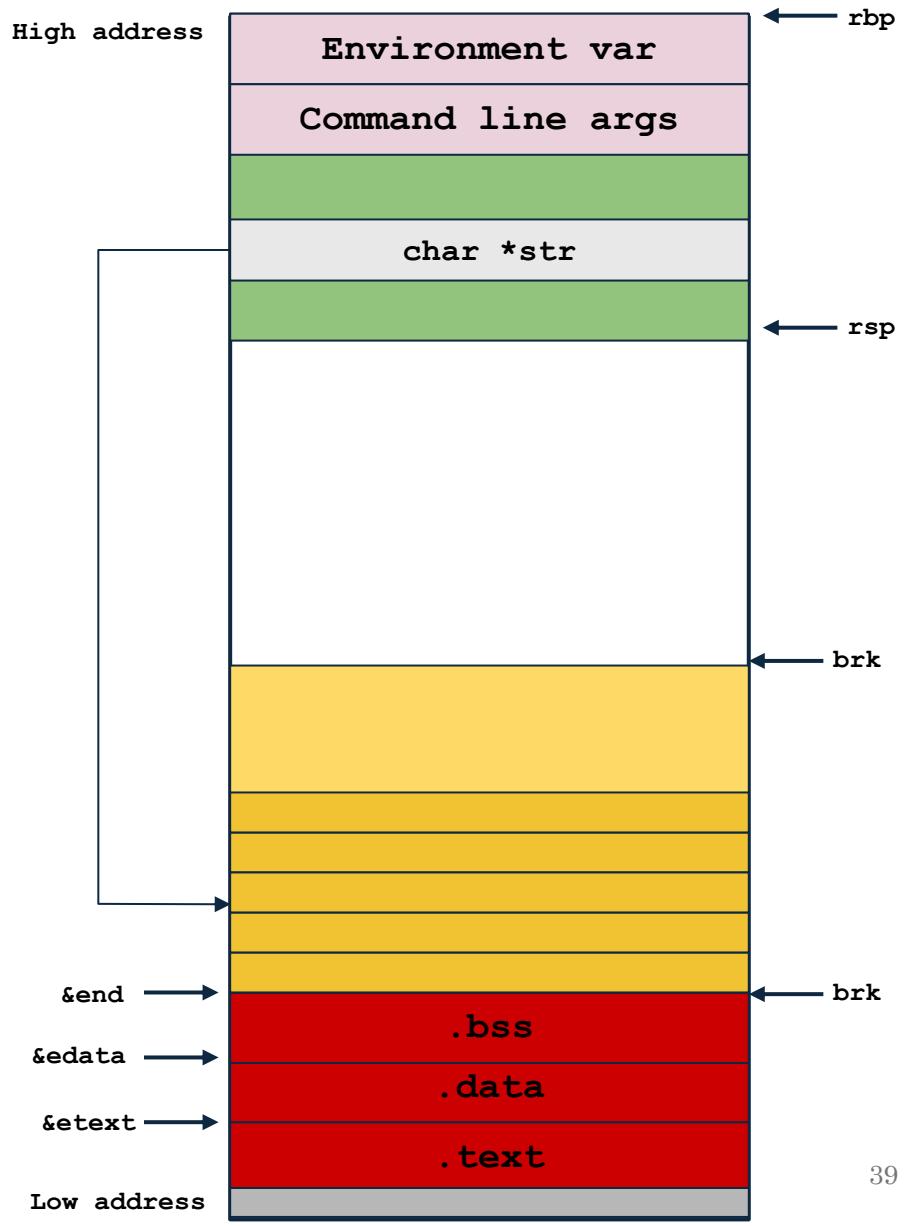
```
void free ( void* ptr );
```

- **`free()`** deallocates the block of memory pointed to by its pointer argument, which should be an address previously returned by functions of malloc package.

Allocating 1D Array on Heap

```
char *str = (char*)malloc(sizeof(char)*10);


…
…
…
free(str);
```



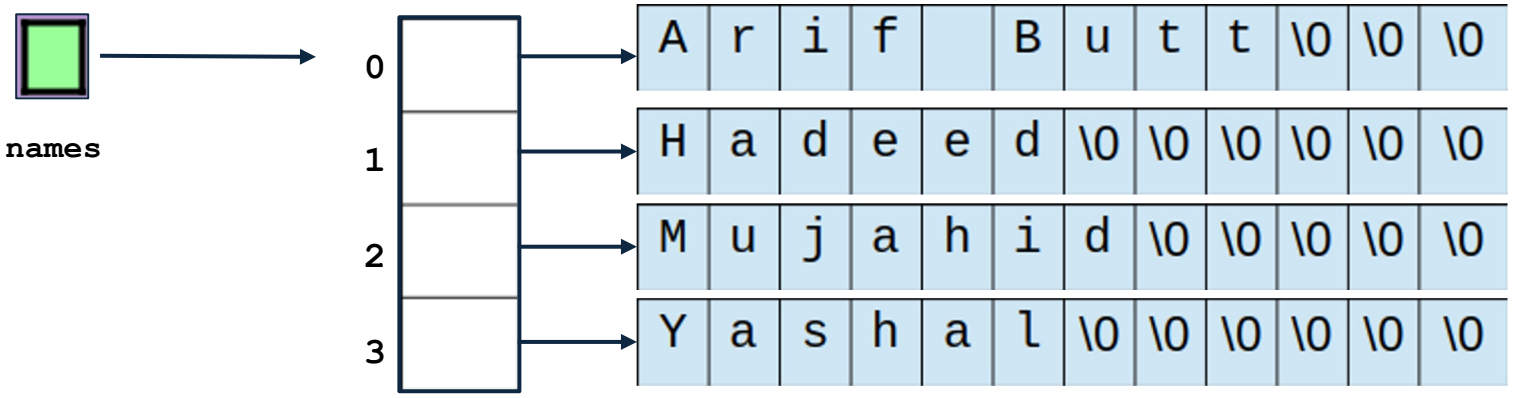
Allocating 2D Array on Heap

```
int i, int rows = 4, cols = 12;
char ** names = (char**)malloc(sizeof(char*) * rows);

for(i = 0; i < rows; i++)
    names[i] = (char*)malloc(sizeof(char) * cols);


for(i = 0; i < rows; i++)
    free(names[i]);

free(names);
```



Demonstration

$100 QUESTION

If a process continuously calls `malloc()`, without calling `free()`, what happens and why?

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Heap: Behind the curtain

System call: `brk()`

```
int brk(void* end_data_segment);
```

- Resizing the heap is actually telling the kernel to adjust the process's program break, which lies initially just above the end of the uninitialized data segment (i.e `end` variable).
- The `brk()` is a system call that sets the program break to location specified by `end_data_segment`. Since virtual memory is allocated in pages, this request is rounded up to the page boundary. Any attempt to lower the program break than `end` results in segmentation fault.
- The upper limit to which the program break can be set depends on range of factors like:
  - Process resource limit for size of data segment.
  - Location of memory mappings, shared memory segment and shared libraries.

Library call: `sbrk()`

```
void *sbrk (intptr_t increment);
```

- `sbrk()` is a C library wrapper implemented on top of `brk()`. It increments the program break by increment bytes.
- On success, `sbrk()` returns a pointer to the end of the heap before `sbrk()` was called, i.e., a pointer to the start of new area.
- So calling `sbrk(0)` returns the current setting of the program break without changing it.
- On failure -1 is returned with `errno` set to `ENOMEM`.

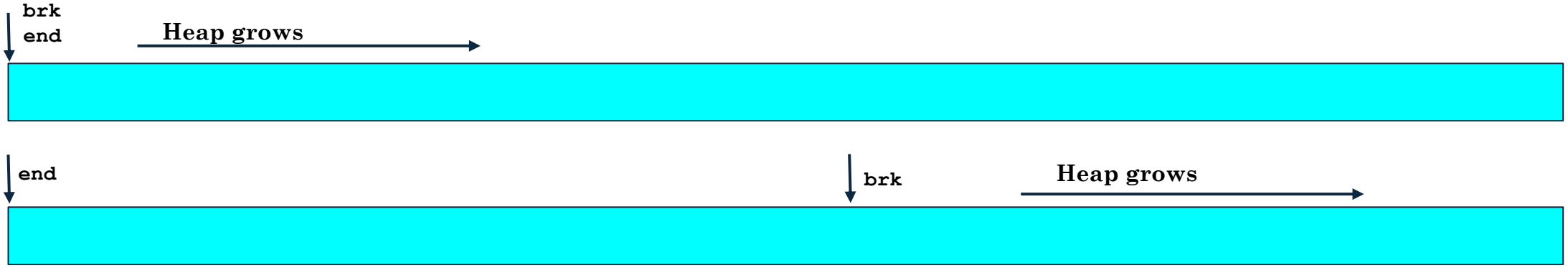
Demonstration

$100 QUESTION

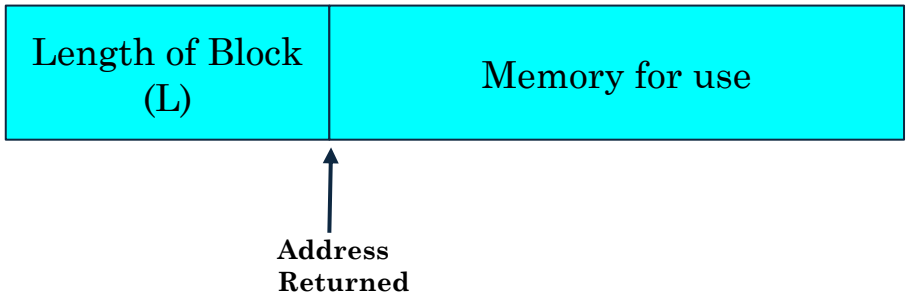
After a process calls `malloc()`, which in turn calls `brk()`, what is the new location of program break? Does it change with every call to `malloc()`?

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

A basic heap allocator



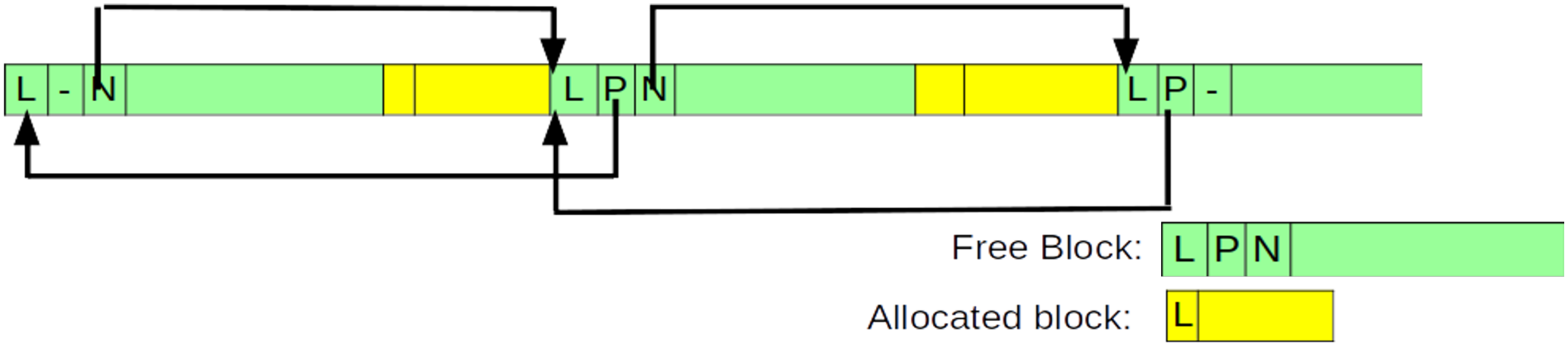
Structure of allocated block on heap



Structure of Free Block on Heap:

Length of Block (L)	Pointer to previous free block (P)	Pointer to next free block (N)	Remaining Bytes of free block

Library call: sbrk()

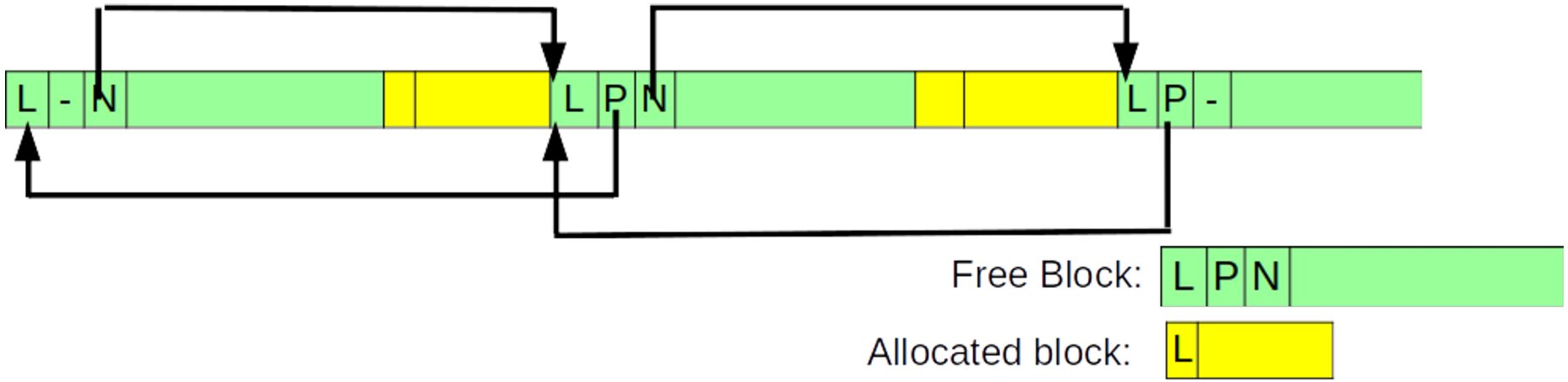


- When a program calls `malloc`, the allocator scans the link list of free memory blocks as per one of the contiguous memory allocation algorithm's (first fit, best fit, next fit), assigns the block and update the data structures
- If no block on the free list is large enough, then `malloc()` calls `sbrk()` to allocate more memory.
- To reduce the number of calls to `sbrk()`, `malloc()` do not allocate exact number of bytes required rather increase the program break in large units (some multiples of virtual memory page size) and putting the excess memory onto the free list.

$100 QUESTION

When a process calls `free()`, how does it know as to how much memory it needs to free? Does it has any effect on program break?

Coalescing Freed Memory



- When you call `free()`, you put a chunk of memory back on the free list.
- There may be times when the chunk immediately before it in memory, and/or the chunk immediately after it in memory are also free.
- If so, it make sense to try to merge the free chunks into one free chunk, rather than having three contiguous free chunks on the free list.
- This is called "**coalescing**" free chunks.

Why not use `brk()` and `sbrk()`

C program use malloc family of functions to allocate & deallocate memory on the heap instead of `brk()` & `sbrk()`, because:

- `malloc` functions are standardized as part of C language
- `malloc` functions are easier to use in threaded programs
- `malloc` functions provide a simple interface that allows memory to be allocated in small units
- `malloc` functions allow us to deallocate blocks of memory

Why `free()` doesn't lower the program break? rather adds the block of memory to a lists of free blocks to be used by future calls to `malloc()`. This is done for following reasons:

- Block of memory being freed is somewhere in the middle of the heap, rather than at the end, so lowering the program break is not possible
- It minimizes the numbers of `sbrk()` calls that the program must perform

Demonstration

$100 QUESTION

	Request Range	→	Actual Chunk Size	→	User Data Available
When a process requests 1-24 B on heap, why the memory allocated is 32 B?	1-24 bytes	→	32 bytes	→	24 bytes
When a process requests 25-40 B on heap, why the memory allocated is 48 B?	25-40 bytes	→	48 bytes	→	40 bytes
When a process requests 41-56 B on heap, why the memory allocated is 64 B?	41-56 bytes	→	64 bytes	→	56 bytes
When a process requests 57-72 B on heap, why the memory allocated is 80 B?	57-72 bytes	→	80 bytes	→	72 bytes

Minimum user data allocation is 24 Bytes with 8-byte header (on 64-bit) containing metadata like size and flags, so a total chunk of 32 bytes.
All requests greater than 24 bytes are aligned to 16-bytes boundaries for optimal CPU performance

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Typical Heap Misuses and Errors

- **Memory Leaks:** Failing to call `free()` for dynamically allocated memory, causing gradual memory exhaustion.
- **Use-After-Free:** Accessing memory through pointers after the memory has been freed via `free()`. The freed memory may be reallocated for different purposes, leading to data corruption, information disclosure, or arbitrary code execution when exploited through heap spraying (an exploitation technique that involves manipulating heap layout through controlled allocations to position malicious data at predictable addresses).
- **Double Free:** Calling `free()` multiple times on the same memory address. This corrupts heap metadata structures, potentially causing heap corruption, crashes, or exploitable conditions where attackers can manipulate heap layout for arbitrary write primitives.
- **Heap Buffer Over/Underflow:** Writing beyond the boundaries of a heap-allocated buffer, corrupting adjacent heap chunks, their metadata, or other allocated objects. Unlike stack overflows, these often target heap management structures like chunk headers, free lists, or bin pointers, allowing exploitation techniques like tcache poisoning.
- **Invalid Free Operations:** Attempting to free memory using pointers that weren't returned by `malloc()`, `calloc()`, or `realloc()`.
- **Uninitialized Heap Memory Use:** Reading from newly allocated memory before initialization. Unlike stack memory, heap memory may contain sensitive data from previous allocations, creating information disclosure vulnerabilities.

These vulnerabilities are mitigated through modern heap implementations (ASLR, heap canaries, safe unlinking), memory-safe alternatives, static analysis tools, and runtime

Tools and Libraries for malloc debugging

Enhancing Heap Debugging with Specialized Libraries

- C lacks built-in memory safety, making programs vulnerable to crashes, security exploits, and unpredictable behavior from memory-related bugs. An early detection prevents costly production issues and security vulnerabilities.
- Use the following tools during during development/testing for comprehensive error detection, and disable in production builds to eliminate runtime overhead and maintain optimal performance.
  - **Valgrind:** Dynamic analysis tool that detects memory leaks, buffer overflows, use-after-free, and uninitialized memory access by running programs in a virtual machine environment.
  - **Electric Fence:** Memory debugging library that detects buffer overflows by placing inaccessible guard pages immediately before and after each heap allocation. (`gcc -g -o program program.c -lefence`)
  - **AddressSanitizer (ASan):** Fast memory error detector built into GCC that instruments code at compile-time to catch buffer overflows, use-after-free, and memory leaks with minimal runtime overhead. (`gcc -fsanitize=address -g -o program program.c`)
  - **Use gcc Static Analysis option:** Compile-time static analysis examines source code without execution to identify potential bugs, memory leaks, and security vulnerabilities. (`gcc -fanalyzer -Wall -Wextra -o program program.c`)

Valgrind Debugging Framework

Valgrind is a modular instrumentation framework, providing dynamic analysis tools for memory, threading, cache and heap issues:

- **Memcheck:** Default memory-error detector (detects invalid access, uninitialized memory, leaks, double-free)
- **Cachegrind:** Simulates CPU caches to analyze cache misses and memory references per line/function.
- **Callgrind:** Call Graph profiler.
- **Helgrind:** Detects threading errors like data races in POSIX threads.

While using `valgrind`, it is recommended to compile your program with options like `-g` (debug symbols) and preferably avoid high optimization levels like `-O2` to avoid false positives (`-O0` is recommended for Memcheck). The general usage syntax is:

```
$ valgrind [--tool=toolname] [tool-specific options] yourprogram [program options]
$ gcc -O0 -g faultyprog.c -o faultyprogram
$ valgrind --tool=memcheck ./faultyprogram
$ valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all ./faultyprogram
```

Example: faultyprogram.c

```
void f1();
void f2();
void f3();
void f4();
void f5();
void f6();
void f7();
int main(){
    f1();
    f2();
    f3();
    f4();
    f5();
    f6();
    f7();
    return 0;
}
```

```
void f1(){
        int number = 54;
        if (number = 3)
           printf("variable contains 3\n");
        else
           printf("variable do not contain 3\n");
}
void f2(){
        char * ptr = (char*) malloc(sizeof(char)*10);
        ptr[13] = 'a';
        free(ptr);
        printf("Bye Bye from f2()...\n");
}
void f3(){
    char * ptr = (char*) malloc(sizeof(char)*10);
    free(ptr);
    ptr[5] = 'a';
    printf("Bye Bye from f3()...\n");
}
```

- The `f1()` function uses **assignment** (=) instead of **comparison** (==) in the `if` statement. → Valgrind won't catch assignment-vs-comparison logic errors.
- The `f2()` function **writes beyond allocated memory** → The Memcheck will report Invalid write in `f2()` and points to the exact line, so you can fix the allocation size or remove invalid access.
- The `f3()` has **use-after-free** vulnerability, as it tries to write to memory after it has been freed → Memcheck will detect access to freed memory specifying "Invalid read/write after free"), prompting removal of such operations.

Example: faultyprogram.c (cont…)

```
void f1();
void f2();
void f3();
void f4();
void f5();
void f6();
void f7();
int main(){
    f1();
    f2();
    f3();
    f4();
    f5();
    f6();
    f7();
    f8()
    return 0;
}
```

```
void f4(){
    char * ptr = (char*) malloc(sizeof(char)*10);
    free(ptr);
    free(ptr);
}
void f5(){
    char * ptr = (char*) malloc(sizeof(char)*10);
    ptr[1] = 'a';
    printf("Bye Bye from f5()...\n");
}
void f6() {
     char buffer[5];
     int secret = 12345; // should remain unchanged
     printf("secret = %d\n", secret);
    for (int i = 0; i <= 8; i++)
         buffer[i] = 'A' + i;
     printf("secret = %d\n", secret); //134770388
}
```

- The `f4()` function has double free vulnerability, as it calls **free twice** on the same pointer. → The Memcheck will report "Invalid free()", so you can ensure pointers are set to NULL after freeing.

- The `f5()` function suffers with **memory leak** vulnerability as the allocated memory is never freed. → The Memcheck will report "definitely lost" blocks with size and location, guiding you to add proper `free()` calls.

- The `f6()` function suffers with **BoF vulnerability** as it writes 9 chars into a 5-sized array and overwrites the 4-byte secret number. → The Memcheck will report "Invalid write of size 1".

Example: faultyprogram.c (cont…)

```
void f1();
void f2();
void f3();
void f4();
void f5();
void f6();
void f7();
int main(){
    f1();
    f2();
    f3();
    f4();
    f5();
    f6();
    f7();
    return 0;
}
```

```
char* getString1(){
    char msg[100] = "Finding bugs is fun with Arif";
    char *ret = msg;
    return ret;
}
void f7(){
    char * abc = getString1();
    printf("String: %s\n",abc);
}
char* getString2(){
    char * ret = (char*) malloc(100);
    strcpy(ret, "Finding bugs is fun with Arif");
    return ret;
}
void f8(){
    char * abc = getString2();
    printf("String: %s\n",abc);
    free(abc);
}
```

- The `f7()` function calls `getString1()` function which **returns a pointer to a local stack variable** named `msg`, causing undefined behaviour when accessed. → Although not strictly a heap issue, Valgrind can indirectly expose this, when it reports invalid reads from freed stack frames.

- The `f8()` function calls `getString2()` function, which returns a pointer to an unnamed heap memory (good practice). However, it appears that the `getString2()`, function relies on the caller to free memory, which is of course a good practice.

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

To Do

- Watch SP video on Process stack:

  https://youtu.be/1XbTmmWxHzo?si=PkqxEbnWP2LIPqYZ

- Watch SP video on Process heap:

  https://youtu.be/zpcPS27ZQr0?si=OWJyNUJV_Lo-sLYj

- Watch Basic Programming C video dynamic programming:

  https://www.youtube.com/watch?v=EK1L2wYz_iE

Coming to office hours does NOT mean that you are academically weak!